JOURNAL OF APPLIED
COMPUTER SCIENCE
Vol. 18 No. 2 (2010), pp. 25-55

On Distributed Data Processing in Data Grid Architecture for a Virtual Repository*

Kamil Kuliberda[1], Jacek Wiślicki[1], Tomasz M. Kowalski[1], Radosław Adamus[1], Krzysztof Kaczmarski[2], Kazimierz Subieta[3]

[1]*Technical University of Łódź, Computer Engineering Department*
Stefanowskiego 18/22, 90-924 Łódź, Poland
{kamil, jacenty, tkowals, radamus}@kis.p.lodz.pl

[2]*Warsaw University of Technology*
Faculty of Mathematics and Information Science
Plac Politechniki 1, 00-661 Warsaw, Poland
kaczmars@mini.pw.edu.pl

[3]*Polish-Japanese Institute of Information Technology*
Koszykowa 86, 02-008 Warsaw, Poland
subieta@pjwstk.edu.pl

Abstract. *The article describes the problem of integration of distributed, heterogeneous and fragmented collections of data with application of the virtual repository and the data grid concept. The technology involves: wrappers enveloping external resources, a virtual network (based on the peer-to-peer technology) responsible for integration of data into one global schema and a distributed index for speeding-up data retrieval. Authors present a method for obtaining data from heterogeneously structured external databases and then a procedure of integration the data to one, commonly available,*

*This research work is funded from the science finances in years 2010/2012 as a research project nr N N516 423438.

global schema. The core of the described solution is based on the Stack-Based Query Language (SBQL) and virtual updatable SBQL views. The system transport and indexing layer is based on the P2P architecture.
Keywords: *virtual repository, wrapper, object to relational mapping, P2P, distributed index, grid integration.*

1. Introduction

A grid is a novel technology widely researched by many academic and industrial organizations. Recently, every year there are organized more than 10 different grid-oriented conferences in the world. The grid-related researches are very important not only in computer science but also in business area. There are lot of different approaches and solutions for grid realizations for different resources and processes.

Formerly a grid was referred mostly to computational networks, however, due to the rapid evolution of the Internet technologies and increase of a worldwide business information exchange, there have arisen further expectations towards grid systems. Well known P2P systems, where communities manage a flat data, are not sufficient in contemporary business applications. Thus, people want to reach higher forms of information processing and integration a structured data. Such data come mainly from database resources. These new opportunities have opened doors for our proposal which deals with a distributed parallel database content, where various data and service resources residing in separate locations can be virtually available through their global representation. This technology is referred to as a data-intensive grid or just a data grid. Such a global representation (a view) should abstract users from all the technical aspects of the process of a data integration (which concerns location, heterogeneity, fragmentation, replication, redundancy, etc.), and which is referred to a transparency. The effect of the transparency requires enveloping (wrapping) the grid resources with dedicated programmatic structures - wrappers.

The paper is focused on the data grid (DG) architecture devoted to data-intensive applications. It covers technical aspects of data processing in the virtual repository (VR) and try to describe a complete architecture for transparent processing of different data formats. The proposal rely on the Stack Based Approach (SBA) and the query/programming language SBQL (based on SBA). The VR platform implementation is based on the prototype od an object oriented database environ-

ment ODRA. [20]. The main VR components includes a peer-to-peer transport platform (TP) as a non-limited communication middle-layer, a global index for resources indexation and an object-to-relational wrapper for sharing legacy data in the VR environment. In our opinion this architecture is well suited to many situations where distributed, heterogeneous and redundant data resources that are to be virtually integrated into a centralized, homogeneous and non-redundant whole. The important element of the VR architecture is an distributed index (DI) for indexing distributed data. The VR deals with some aspects of higher forms of distribution transparency and offers some common infrastructures build on top of the grid, including the trust infrastructure (security, privacy, licensing, payments), web services, distributed transactions, workflow management, etc.

Contemporarily, there are a lot of solutions where various forms of data from distributed resources become unified into one VR with a common data schema. Such approaches are popular in database environments and they let users to achieve many forms of transparent accesses to heterogeneous resources. In such solutions a user is not aware of an actual data form as he or she gets only needed information in the best shape for a particular use. Among many other new concepts in modern databases this branch evolves and develops very quickly as an attractive answer to business needs. There are many potential applications of dynamic data integration technologies in a modern society, where data must be accessible from anywhere, at any time, for example e-Government, e-University or e-Hospital. Common technological approaches involve a semantic data description and an ontology usage extended by logic-based programs that try to understand user needs, collect data and transform it to a desired form (RDF, RDFQL, OWL). Other commercial systems like Oracle-10G offer a flexible execution of distributed queries but they are still limited by a data model and languages not sufficient for distributed queries, in which programming suffers from inflexibility, complexity and many unpredictable complications. There are also several open technologies designed to share or exchange data like Edutella [10], OGSA-DAI [11] and Piazza [17], [18].

In this article we present a data integration environment providing query-programming language facilitates that, in our opinion, are able to express very complex integration scenarios (including updates), and still keep programming model simple and easy to maintain. The rest of the paper is organized as follows. In Section 2 we present basic functional elements of VR and DG. In Section 2 subsections there are discussed details of SBA and ODRA implementations, distributed data representation inside VR with DI mechanism. There is also description of TP and GI components with an appropriate integration data process. At the end of this sec-

tion there is an introduction to the generic wrapper architecture. Section 3 presents in details an example how we integrate distributed resources. Section 4 concludes.

2. Details of a Virtual Repository and its Data Grid Mechanism

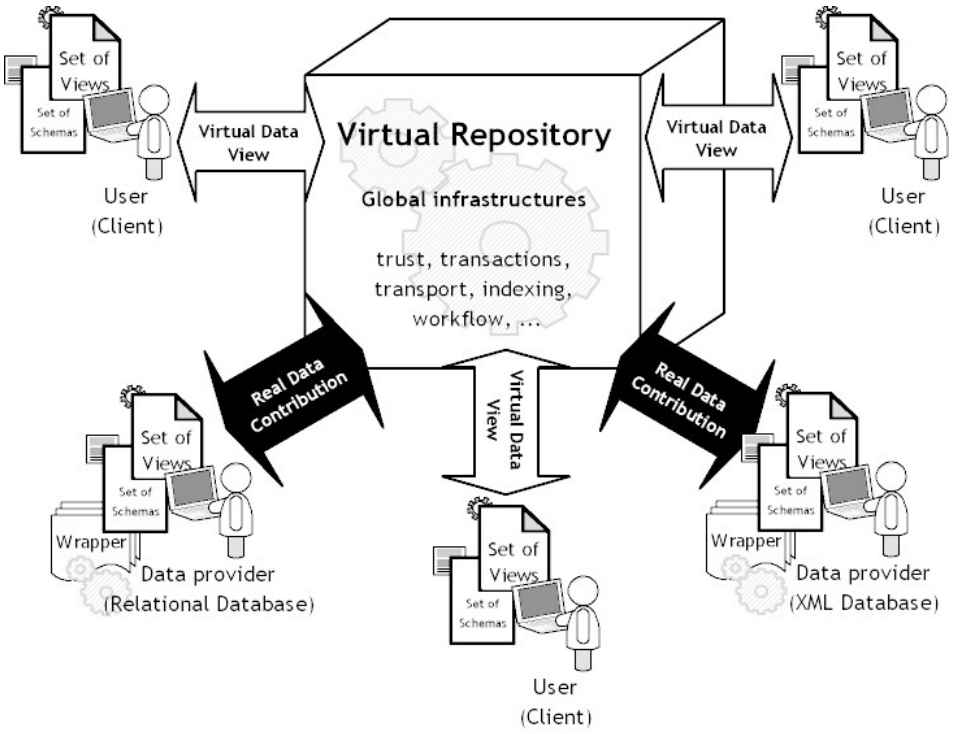

Figure 1. The concept of a Virtual Repository. Users work with their own and favorite view of resources not knowing the real data or services

The solutions exploiting virtual repositories (VR), currently available in the field, implement particular approaches which are hard to reuse for other organizations and business goals. Processing in the VR comprises some complex issues. One of them is updating virtual data seen through the VR. The state of research on this problem is open, practically unexplored. Similar problems concern security

and global transaction infrastructures which are built over of the VR. Another one concerns performance issues, in particular, a global query optimization concerning our object oriented database query language as a user interface for the VR.

The problems described above lead to an idea of developing generic methodologies, environments, tools and languages that support quick development of a grid with the virtual repository aiming at a particular application integration goal. This problem can be solved within an architectural idea of the grid components presented in Figure 1. This also reflects on a developing of concrete technical solutions concerning particular components:

- developing a canonical model and a schema according to global user requirements (several directives like a business contract, a standard, law regulations, etc.). The model and the schema should be implemented in a corresponding language having both human and machine interpretations,

- developing local models and schemata of providers for participating local data and service resources in terms of a canonical model and a schema, i.e. showing how particular local providers contribute to the global schema,

- developing an integration schema exploiting a local schema and a global schema which shows dependencies between local resource providers and the global view and the dependencies between the local providers (redundancies, replications, etc.),

- developing a communication middle-layer which performs on easy connection of VR participants and where data may be transported without any limitation (e.g. NATs, firewalls, etc.) between all connected units. This module also must assure keeping the VR and its contents up-to-date,

- developing a tool for treatment remote objects, it must create a transparent process of catching the remote objects and make them available for local users like their local objects,

Although, the literature contains many works concerning the above issues and problems, the field is rather in a premature stage, far from a complex and universal solution. Our research based on generic object-oriented database model with updatable views focused on unifying query languages and updatable views to design well defined grid mechanisms creates a big chance to receive significant theoretical and practical results much beyond the current state of art.

The technical aspects of realization of the described idea assume an existence of several cooperating technologies nearby. The principal aspect is a good design of a virtual repository platform corresponding with two other aspects which are responsible for data resources maintenance (wrappers) and a business data exchange

including a grid security and its management (a TP). The concept concerns existing individual software modules with interconnectivity mechanisms enabling a restricted and specified participation of data processing. It has an original and general tendency. The modules may be developed and implemented according to the idea of a project's architect.

Our concept assumes the following strategies:

- a virtual repository (VR) - physically available with applications based on the previously described architecture. Parts of a repository will be client, provider application and management application.

- a transport platform (TP) [6] - determines independent software environment responsible for free distributed transaction processing. The platform particularly should grant an unlimited physical access to a grid network for clients and resource providers (units) and an assurance of a well formed protocol for an information interchange. It is based on a centrally managed peer-to-peer network infrastructure. The following P2P features should assure operations such as: unit unique identifying, unit naming, units' interconnections, network security, etc. Other important aspects keep resources location transparent for acting units, a scalability of a network, an independence of a physical network configuration and naming. All these aspects can be developed using a multiprotocol, fully programmable P2P platform of the JXTA project [13].

- import/export adapters and wrappers - mechanisms supporting a grid architecture to import and export local resources which may contribute to a virtual repository. They are software modules enabling a resource provider's services exploitation. Each of grid clients and providers will be equipped with, a user selected or corresponding to a grid, contribution schema module. Such module can discover local data and by the views mechanisms grant access as a part of a virtual repository [4].

- developing assumptions concerning export wrappers for particular providers sharing their resources.

2.1. SBA and SBQL in ODRA Platform

Integration of the distributed data requires sophisticated and flexible platform with an adequate set of features including:

- generic data model that is able to express wide range of possible models of data sources,

- powerful query/programming language with full computational power of pro-

gramming languages seamlessly integrated with a high-level query languages capabilities,

- virtual views with full update functionality that allow to: wrap local resources into a common data model; build global applications through integration of local resources; customize global assets to the needs of a particular client applications.

All the above mentioned features are hardly to achieve using current state-of-the-art platforms, programming and/or query languages and common data models. Fortunately the methodology called Stack Based Approach (SBA) to query-programming languages [14], [15] is able to equip the SBA-based platforms with all the tools to work out required solution. SBA extends the database programming with all the popular object-oriented mechanism and introduces new unknown previously, like e.g. dynamic object roles and virtual updatable views.

The query language, which is based on SBA, is called Stack Based Query Language (SBQL) [14], [15]. It is new self-contained query/programming language without distinction between traditional programming language expressions and declarative constructs (queries) for data processing. SBQL is defined on very general data store model based on object relativism and full internal identification principles.

SBQL also support imperative constructs and mechanisms, e.g.: control structures, procedures, classes, interfaces, modules. One of the most important features of SBQL are virtual updatable object views [19]. The entirely new property of SBQL views is their full updatability. View designer/programmer has full control over the update operations that are performed on the virtual objects defined by the view. There are no restrictions concerning the type of allowed operation. The view definition can contain nested view definition, procedures, variables, etc.

The advantages of SBA and SBQL have been implemented in a project called ODRA (Object Database for Rapid Application development) [20]. The aim of the ODRA is to design an object-oriented application development platform - the tool for future database application programmers. Because the ODRA features are still under development and continuous extension, below we summarize only the most important ones:

- core of the ODRA environment is the SBQL language defined for a general data store model based on the object relativism principle. There are no dangling pointers and null values,

- queries are treat in the same way as expressions in popular programming languages. The design is based on the compositionality principle (e.g. select-from-where syntactic sugar is avoided),

- names occurring in queries are bound using environment stack (ENVS), a structure well known from popular programming languages implementations,

- operators are divided for the sake of its association with the ENVS: algebraic (e.g. +, -, auxiliary name) do not use ENVS; and non-algebraic (navigation, where, quantifiers, join, etc.) which use it in a similar way to procedures (opens new ENVS section and execute against new ENVS state),

- ODRA introduces the concept of updatable object views that allows performing a transparent update operation on a virtual data. The views are the core for data and application integration using ODRA [4],

- ODRA implementation includes strong query optimization techniques based on query rewrite (procedures and views rewrite), query modification (independent sub-queries, removing dead sub-queries) [21] and indices,

- For execution of SBQL programme the ODRA provides virtual execution mechanism called Juliet. The Juliet virtual machine consists of bytecode, bytecode interpreter and set of services (virtual memory, scheduling, loading, security),

- ODRA is able to plug an existing data sources through wrappers (e.g. relational data) and filters (e.g. XML and RDF). Next, such a data can be queried with SBQL queries.

All the above mentioned features of the ODRA environment predestine the platform to be an excellent tool for implementation of virtual repositories for data grid applications.

2.2. Distributed Data Representation through Data Grid in Virtual Repository

The proposal for processing distributed and heterogeneous resources presented in previous subchapter assumes different approach to work with data than similar solutions like [10], [11], [16], [17]. Our solution represents a Data Grid architecture which is described in details in [7]. We claim that neither data nor services can be copied, replicated and maintained in the centralized server. They are to be supplied, stored, processed and maintained on their autonomous sites [7], [8]. The external resources should be easily pluggable into the system as well as users can appear and disappear unexpectedly.

A user as well as a data provider (see Figure 1) may plug into a VR and use its resources according to his or her requirements, availability of the resources and assigned privileges. The goal of our research is to design a platform where all users and providers are able to access multiple distributed resources and work on

the ground of a global schema for all the accessible data and services. A virtual repository should present a middleware supplying a fully transparent access to distributed, heterogeneous and fragmented resources from its clients [1], [5], [6].

Looking on the system at its participants' side (clients and data providers) there are two kinds of data schemata. The first, a contributory schema - it is description of a local resource acceptable for the virtual repository. A virtual repository can deal only with the data which is exported by the decision of a local administrator. Another reason for limited access to local resources is a consortium agreement, which is established for the VR. The agreement has certain business goals and not need to accept any data from any provider [5], [6]. As the second schema we claim, is a description of global data and services available for clients. Such schema is named global user schema.

The basic assignment to solve for the VR is transformation of local client's data through contributory schemata into a global user schema. The transformation can perform more sophisticated homogenization of data and integration of fragmented collections. This is done by updatable views which are able to perform any data transformation and support view updates without limitations which are commonly known from other similar systems. Our views have the full algorithmic power of programming languages, thus they are much more powerful than e.g. SQL views.

A responsibility of management of grid contents through access permissions, discovering data and resources, controlling location of resources, and indexing all grid attributes are basic tasks of a global infrastructure. The design and implementation challenge is a method of combining and enabling non-limited both-way processing of clients' and data providers' contents which participate in VR's global virtual store [8].

Our DG architecture provides a views system (Set of Views in Figure 1) available to every VR's participant. Moreover, each data provider possesses a view which transforms its local share into an acceptable contribution, a contributory view. Providers may also use extended wrappers to existing DBMS systems (see sub-chapter 2.7 and [6], [7]). Similarly, a client uses an integration view to consume needed resources in a form acceptable for his or her applications. This view is performing the main task of data transformation and its designer must be aware of data fragmentation, replication, redundancies, etc. [3], [5]. This transformation may be described by an integration schema prepared by business experts or being a result of automatic semantic-based analysis. The problem is how to allow transparent plugging in new resources and how to incorporate them into existing and working views. This question is discussed in section 2.4.

2.3. The Distributed Index for Virtual Repository

Indices are auxiliary (redundant) database structures stored at a server side. A database administrator manages a pool of indices generating new or removing them depending on the need. As indices in the end of the book are used for quick page finding, database indices quicken retrieving objects (or records) matching given criteria. The advantage of indices is their relatively small size (comparing to a whole database) and a single aspect search, which makes their organization very efficient.

Implementing indexing in an object-oriented data grid requires applying standard local indexing techniques and other techniques dedicated to a distributed database environment. This issue is still not deeply studied and requires introducing new solutions. Authors focus on Scalable Distributed Data Structure (SDDS) [9] as a basis for a distributed index organization in order to optimally utilize data grid computational resources.

A query optimization schema for local and distributed indexing of data which considers an index transparency is under development and implementation in the ODRA prototype. This aspect should be presented in a separate paper due to its complexity and introduced solutions. So far in the proposal [3] a grid stores just server link objects. However, it could also take care of indices. Building global indices kept by a virtual store has very significant potential for an optimization of grid computing. An idle time of a global virtual store can be filled with indexing and cataloguing data held by local servers.

For users a grid technology should satisfy following general requirements: a transparency, a security, an interoperability, an efficiency and a pragmatic universality.

The most important for a complexity of a design, programming and a maintenance effort addressing distributed data and services is a transparency. It reduces a complexity of a global application. One of forms of a transparency is in a field of indexing. As in relational databases, programmer must not to be aware of indices that are introduced to improve performance, because they are used automatically during a query evaluation. Therefore, an administrator of a database can freely generate new indices and remove them without need to change a code of applications.

Knowledge about indices existing in local stores is not and does not need to be available on a level of a global schema. A grid user query, during a process of rewriting optimization, in many cases can be decomposed into sub-queries sent to

particular grid participants. Such a sub-query concerns data stored locally on a target peer and does not take advantage of existing local indices. Before an evaluation a grid participant can optimize it by rewriting according to a local metabase and a local index repository. There are many advantages of local indexing strategy in a distributed environment:

- data and indices are localized on the same server so only local methods for preserving an indices cohesion are needed,

- global query optimization is divided between grid peers on global and local levels. A global optimizer does not have to take into account local optimizations.

Local indexing is transparent to a grid. Local indexing is not always sufficient regarding a computational power of a grid, however we purpose using distributed index data-structure. SDDS is a scalable distributed data structure introduced by [9] which deals with storing index positions in a file distributed over a given network. Its properties make it a good candidate for indexing local or global data in a grid infrastructure. SDDS uses LH* which generalizes a linear hashing technology to a distributed memory or disk files. An application of SDDS introduces following features in distributed indexing:

- avoids a central address calculus spot,
- supports a parallel and distributed query evaluation,
- provides a concurrency transparency,
- scalability - it does not assume any constraints in a size or a capacity,
- SDDS file expands over new servers when an optimal load of current servers is reached,
- index updating does not demand a global refresh on servers or clients,
- over 65 percent of a SDDS file is used,
- in general a small number of messages between servers (1 per random insert; 2 per key search).

With all these characteristic SDDS outperforms in an efficiency of the centralized index directory approach or any static data structures, therefore, it is planned to be implemented in a currently developed grid platform prototype [2], [3], [4].

Integration of distributed resources into grid, as in examples shown in next sections, due to large amount of data to be processed needs optimization. As it was shown in [12] strong optimization methods can be easily introduced in SBA approach. Indexing plays a very important role in optimizing evaluation of integrating queries because it significantly reduces amount of processed data and therefore lightens a grids load.

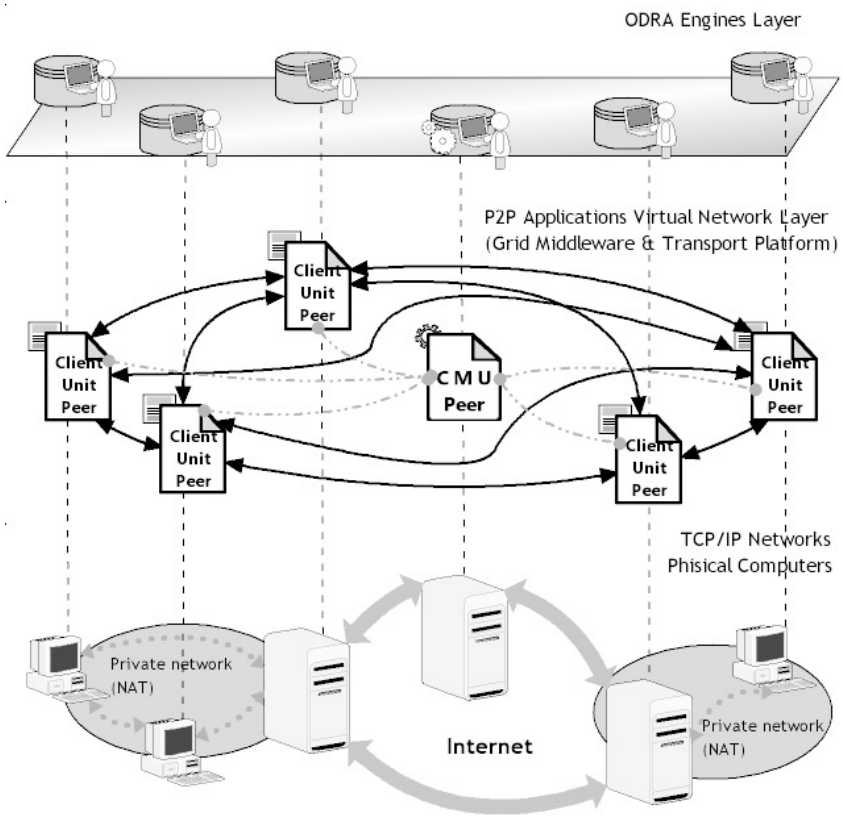

Figure 2. Data grid communication layers and their dependencies

2.4. The Transport Platform for Data Obtaining

A grid oriented communication requires several additional solutions for flexible data transportation between the VR's participants through its virtual network. Especially the transport platform solution should comply following functionalities like; (1) transparent data processing (queries and results), (2) transparent integration of resources, (3) dynamic users joining, (4) rapid indexation of available resources, (5) trust infrastructure for contributing participants. The general architecture of the TP concept solves the above issues through a middleware platform mechanism designed for an easy and scalable integration of a community of database users. It creates an abstraction method for a communication in a grid

community, resulting in a unique and simple database grid, processed in a parallel peer-to-peer (P2P) architecture [6] realized with the JXTA package [13]. Proposed realization of the TP has an additional advantage, the P2P communication is always non-dependent of TCP/IP stack limitations, e.g. firewalls, NAT systems and encapsulated private corporate restrictions. Thus, our grid solution is more flexible for the users than OGSA [11] and Piazza [16], [17] where communication relies on standard client-server TCP/IP protocols. Additionally our solution creates network processes (such as an access to the resources, joining and leaving the grid) that are transparent for the participants.

User grid interfaces - in this proposal database engines - are placed over the P2P network middleware. DBMS-s work as heterogeneous data stores, but in fact they are transparently integrated in the VR through P2P interfaces (separate transparent applications placed in P2P virtual network). Users can process their own local data schemata, but additionally they are able to work on remote business information from global schema available for all contributors. This part of data grid activity is implemented with top-level user applications available through database engines and SBQL query language [14], [15], see OODBMS Engines Layer in Figure 2. In such architecture, databases connected to the virtual network's P2P applications arrange unique parallel communication between physical computers for an unlimited business information exchange.

Our proposed virtual network has a centralized architecture whose crucial element is a central management unit (CMU) showed in Figure 2. There is an assumption that inside the P2P virtual network could be placed only one CMU peer being responsible for a DG's lifetime. Besides, it manages the VR integrity and resource accessibility. At the P2P virtual network level, the CMU has some responsibilities, the most important are; (1) a creation of the grid network, (2) a managing the grid network, (3) a handling errors and status of the grid network.

For the regular VR users - we mean clients and data providers, separate P2P applications are implemented. They are depicted on a Figure 2 as Client Peers. These applications are interfaces to the VR for OODBMS users' engines. In this architecture (see Figure 2) each database has its unique name in local and global schemata which is bound with the P2P application's unique name in the virtual network. If current database ID is stored in the CMU, a user can cooperate with others and process information in the VR (according to a trust infrastructure). This is only possible through the OODBMS engine with a transparent P2P application. Unique peer identifiers are a part of P2P implementation of JXTA platform [13], [18].

A P2P application contains an additional separate embedded protocol for a local communication with OODBMS engine. All exceptions concerning a local database operation, virtual network availability and TCP/IP network state are handled by this protocol and P2P application which is responsible for a local part of grid maintenance. Notice that in one local environment two separate applications (P2P virtual network application and OODBMS engine) reside and compose (in grid aspects) one logical application [6].

2.5. Remote Objects in Distributed Resources

As a result of data processing within distributed database nodes, objects may be exchanged and sent between clients and data providers. Transport platform should not limit this behavior if it is only not forbidden by certain VR agreement regulations. We assume, that an object of any kind (simple or complex data or procedure) may travel from its original server to any other server in the VR using the TP. Such an object living in place other than its origin is called a remote object. Again our system should not limit operations that may be done on remote objects if it is not forbidden by organization's agreement. Such an agreement may say, for example, that a patient object may be read by any application and may be updated by any application which has rights of at least a hospital.

If there are many peers in the system, there are also many views which are used to share and access data. After evaluation of a query on a global level (within responsibilities of the global schema) an object may be exchanged many times before it reaches its destination and thus, its update may go through several servers. To be able to sent any object to any destination and also track its update we must use global object identifiers and path tracking.

Our TP solves this problem in rather classical way. Each object gets a unique identification in the time it is created. Unique global object's id is based on unique identification of its original server. If it is sent to another node its identification part is extended by the identification of the destination node. Subsequent object's exchange may add another parts to its identification. In this way remote object's global identification contains not only its source node and identification within that node, but also all the path it traveled to the destination node. If one wants to update a remote object all the nodes which took part in its travel may be informed about the change and may participate in change propagation.

For example, let us consider two steps of address book data exchange. The first node publishes just names of employers. The second node (first stage of data inte-

gration) adds address information to each of the employee object. The third node (second stage of data integration) adds telephone and email information. If such a complex object is modified in another node, update information may be sent to all the nodes participating in the object construction. What's more important a node responsible for certain part of a remote object may control and verify the update procedure.

2.6. Generic Integration Model for Distributed Resources

The real grid systems may have property of permanently changing resources. Thus, our VR technology must react on these changes and a state of VR must be permanently updated. We claim it to be the most important aspect of the VR operating. For an easy management of the VR's content, we have equipped the CMU with a global index mechanism, which covers all technical networking details, management activities and is also a tool for efficient programming in a dynamically changing environment. The global index not necessarily has to be really centralized as there are many ways to distribute its tasks. However, the system needs this kind of additional control and it does not matter how it is organized. Its tasks are: (1) collecting information about available clients and data providers, (2) keeping information on alive peers, (3) indexing location of available objects, (4) managing information about fragmentation types of the available objects, (4) registering and storing information on available resources, (5) keeping network statistics.

The global index is an object which has a complex structure. Its interior reflects object structure of the VR's global schema. The global index can be accessed with SBQL syntax (as a typical database object) on each database engine plugged into the VR. This means that we can evaluate the queries on its contents. There is however one substantial difference from processing typical virtual repository's objects - as a result of an expression evaluation CMU can return only an actual content of its index, like a location list of on-line grid participants.

The global index is the basic source of knowledge about the content of the VR. Basing on the indexed information, referring the views' system, we can easily integrate any remote data inside the VR. If the data have been indexed already, it can be transparently processed without any additional external interference. Additionally the global index contains objects for characterizing the type of data fragmentation. These objects are dynamically managed through the views' systems whenever the VR contents undergoes a change (e.g. when a resource joins or disconnects). The

global index keeps also dependencies between particular objects (complexity of the objects, etc.) as they are organized in a global schema.

Each indexed object in the global index is equipped with a special sub-object called `HFrag` (for horizontal fragmentation) or `VFrag` (for vertical fragmentation). Each of them keeps a special attribute named `ServerName`, whose content is a remote object - an identifier of a remote data source (see Figure 5). If any new resource appears in a virtual repository, there will be added a suitable ServerName into the global index automatically with appropriate information about it.

Accessing the remote data can be achieved by calling the global index with:
GlobalIndex.Name-of-object-from-global-scheme.(Name-of-subobject).
HFrag-or-VFrag-object.ServerName-object;

Because every change of the virtual repository's content is denoted in the global index, accessing data in this way is the only correct one.

2.7. Wrapping External Data Models into Virtual Repository

A wide presentation of wrapping problems is discussed in previous works [7], [8]. Now, we present some implementation aspects of the mentioned wrapper modules. A grid resource (the ODRA engine) denotes any data resource providing an interface capable of executing SBQL queries and returning SBQL result objects as their results (a local query optimization should be performed also, if possible). A nature of such a resource is irrelevant, as only the mentioned capability is important. In the simplest case, where a resource is an ODRA database, its interface has a direct access to a data store and it is similar to an ODRA database engine (DBMS). However, as our grid aims to integrate existing business resources, whose models are mainly relational ones, an interface becomes much more complicated, as there is no directly available data store - SBQL result objects must be created dynamically basing on results returned from SQL relational queries evaluated directly in a local RDBMS.

Such cases (the most common in our grid's real-life application) force introducing additional middleware, a wrapper showed in Figure 4 as a client-server solution. This kind of architecture was applied for a few reasons. One of them are simplicity of implementation and portability. A standard ODRA database can be extended with as many wrappers as needed (e.g. for relational or semistructured data stores) and plugged into any resource model without any lost of its primary performance.

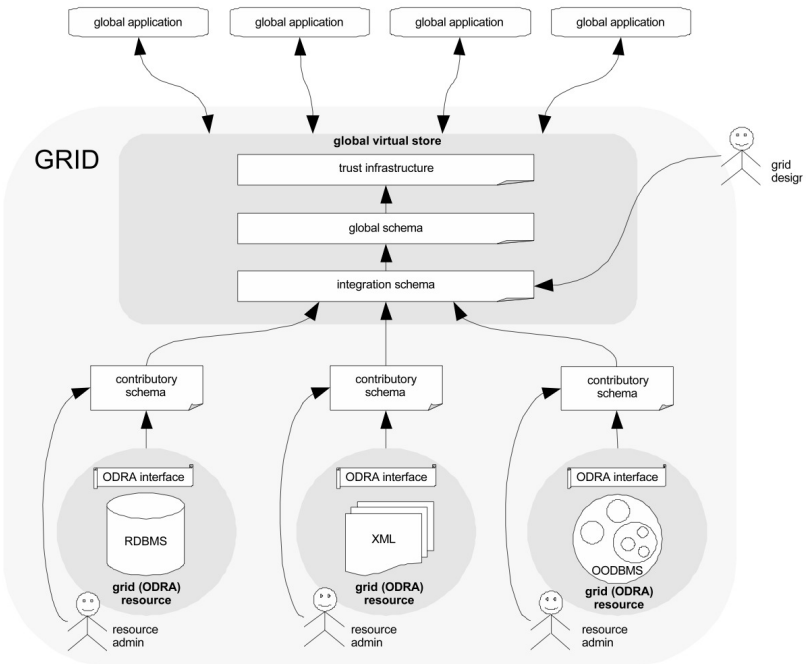

Figure 3. A general architecture of data grid including integration process and wrapper modules

Furthermore, a wrapper server can be developed independently, providing a communication protocol to its client. Of course, an ODRA database with a wrapper's client can work on a separate machine.

A query evaluation process in our data grid environment is depicted in Figure 4. One of the global grid applications sends a query (arrow 1). This query is expressed with SBQL, as it refers to the business object oriented model available to grid users. According to the global schema and its information on data fragmentation, replication and physical location (obtained from integration schemata), the query is sent to appropriate resources. In Figure 4 this stage is realized with arrows 2, 2a and 2b. A notion 'partial query' is general, as in some cases each resource-oriented query can be the same (e.g. in case of a pure horizontal data fragmentation), however in most situations 'partial queries' are different. Query processing corresponding to arrows 2a and 2b is out of the scope of the paper as

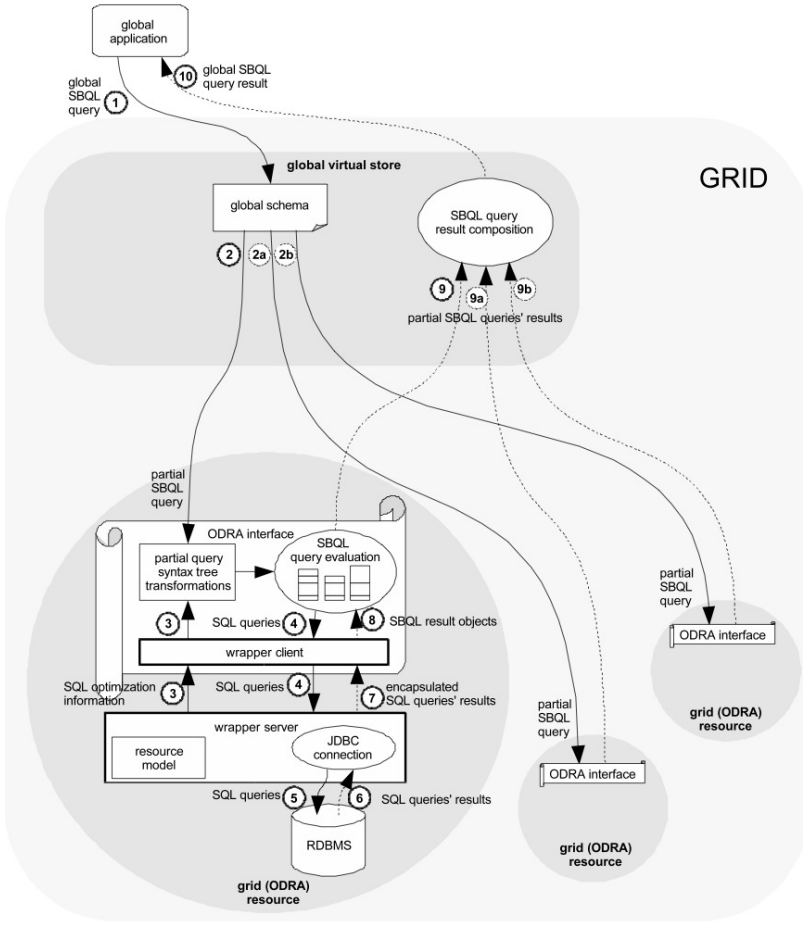

Figure 4. Query evaluation through the wrapper

those grid resources are regarded here as black boxes (any resources conforming with grid requirements, including relational ones described here).

A query evaluation process in our data grid environment is depicted in Figure 4. One of the global grid applications sends a query (arrow 1). This query is expressed with SBQL, as it refers to the business object oriented model available to grid users. According to the global schema and its information on data fragmentation, replication and physical location (obtained from integration schemata),

the query is sent to appropriate resources. In Figure 4 this stage is realized with arrows 2, 2a and 2b. A notion 'partial query' is general, as in some cases each resource-oriented query can be the same (e.g. in case of a pure horizontal data fragmentation), however in most situations 'partial queries' are different. Query processing corresponding to arrows 2a and 2b is out of the scope of the paper as those grid resources are regarded here as black boxes (any resources conforming with grid requirements, including relational ones described here).

The partial query aiming at our relational resource is further processed with a resource's ODRA interface. As mentioned above, the local interface does not have its physical data store, it can only retrieve required data from its RDBMS on-the-fly. First, the interface performs a query optimization. Apart from efficient SBQL optimization rules applied at any grid resource's interface, here we can also transform queries so that powerful native SQL optimizers can work and amounts of data retrieved from the RDBMS are acceptably small. Relational optimization information (indices, cardinalities, primary-foreign key relationships, etc.) is provided by the wrapper server's resource model (arrow 3) and appropriate SBQL query syntax tree transformations are performed. These transformations are based on finding in the tree patterns resembling SQL-optimizable queries. Appropriate tree branches (responsible for such SQL queries) are substituted with calls to `execute immediately` procedures with optimizable SQL queries (their evaluation at the resource is fast and efficient and returned results are acceptably small).

Once syntax tree transformations are finished, the interface starts a regular SBQL query evaluation. Whenever it finds an `execute immediately` procedure, its SQL query is sent to the server via the client (arrows 4, the client passed SQL queries without any modification). The server executes SQL queries as a resource client (JDBC connection), arrow 5, and their results, arrow 6, are encapsulated and sent to the client (arrow 7). Subsequently, the client creates SBQL result objects from results returned from the server (it cannot be accomplished at the resource site, which is another crucial reason for a client-server architecture) and puts them on regular SBQL stacks for further evaluation (arrow 8). In the preferable case (which is not always possible), results returned from the server are supplied with TIDs (tuple identifiers), which enables parameterizing SQL queries within the SBQL syntax tree with intermediate results of SBQL subqueries. Having finished its evaluation, the interface sends it 'partial result' upwards (arrow 9), where it is combined with results returned from other resources (arrows 9a and 9b) and the global query result is composed (depending on fragmentation types, redundancies and replication). This result is returned to the global application (arrow 10).

In a next chapter we present wrapping example for data set which is possible in real-life events.

3. Details of Virtual Repository and Its Data Grid Mechanism

Here we present little complicated example of retrieving particular data objects through our grid solution (VR) for external RDBMS resources which are available through the wrapper modules. At the beginning we introduce how we create and use integration of distributed resources, later we will show an example how we obtain data from external DBMS-s.

3.1. An Integration

Regarding our previous solution defining an integration mechanism for objects fragmented horizontally and vertically (for details see [5]) there was ability to investigate a real life distributed data composition. In this example we deal with different data resources where some of them can store an object with different sub-objects. An object also can store simultaneously identical sub-objects in different resources as a form of data replication. For this purpose a GlobalIndex object was prepared which is also a separate mechanism providing basic information for data integration process. Additionally, it is equipped with HFrag and VFrag subobjects for creating a generic apparatus where all types of data forms and extensions can be addressed.

The situation becomes dramatic if we assume that all of physical data are stored in homogeneous RDBMS-s. This fact forces us to use wrapper modules and catch the data from external DBMS into VR. We will use a full power of our solution to process the data in this exampled grid. First we will show integration procedures for distributed data, after an example on retrieving these data from external resources through wrappers will be shown. All examples use following data structure in the VR (see Figure 5).

The current example shows a method of creating virtual objects from a mixed fragmentation (horizontal and vertical together) including extended dependencies between objects in resources having the same structure, but different data content. This is the most common situation encountered in business data processing, like the following exampled health centre data processing system (Figure 6 and 7).

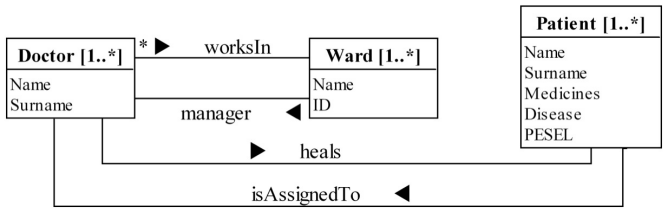

Figure 5. Virtual repository global data schema

Taking an assumption that every health agency has different location and some of them are equipped with a data store containing information of their doctors and wards, then processing must deal with horizontal fragmentation of 'Doctor' and 'Ward' objects. Every health agency must be also equipped with a database of their patients. This is represented by 'Patient' complex objects.

Because these agencies serve special health services according to their specific roles as health institutions (hospitals, clinics, etc.) the situation in data fragmentation becomes more complicated. The patient's personal data should be the same in every health agency, but their treatment history or medicines prescribed are different in every place. Moreover, the system also stores the archival data separately. In this case there must be used a mixed form of a data fragmentation. Health-related patient's data are fragmented vertically on different servers where their personal data, in fact, creates implicit replicas. In this situation we also assume that for each 'Patient' object in a mixed fragmentation contains 'PESEL' subobjects - id number with identical content in every resource. Thus here utilization the knowledge about 'PESEL' content must be employed to make a `join` on fragmented 'Patient' objects. The PESEL attribute is a unique identifier (predicate) for joining distributed objects into a virtual one. Please notice that in the current example we know explicitly which objects are fragmented and how. This situation is depicted in Figure 6, according to this, the content of central index is in Figure 7.

In the example one needs to obtain a list of patients who suffer from "tuberculosis" and who are assigned to doctors working in "cardiology" ward. A query formulated according to specified views definitions should be following:

```
PatientGrid where (Disease = "tuberculosis" and
isAssignedTo.DoctorGrid.worksIn.WardGrid.Name = "cardiology");
```

Basing on solutions presented in [5] defining an integration mechanism for objects fragmented horizontally and vertically a creation of view definition for mixed

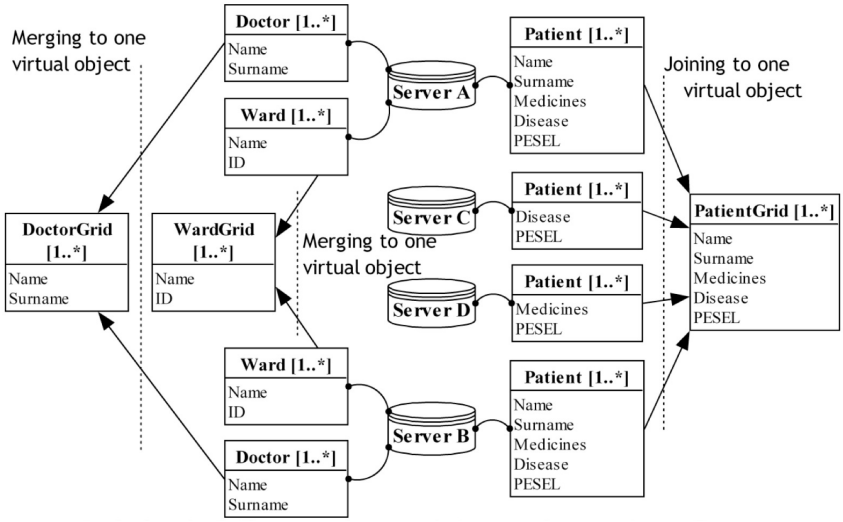

Figure 6. Integration of distributed databases (with mixed object fragmentation) into one virtual structure for a virtual repository

form of fragmentation is rather easy. At first, there must be defined joining procedures for vertical fragmentations in views. After this, the resulting virtual objects must be merged with existing physical objects in a horizontal fragmentation by creating union views' procedures. As a result, this combination of views generates complete virtual objects. In example; the `DoctorGrid` virtual objects definitions (through object views) create virtual objects from horizontal fragmentation. The same situation is for `WardGrid` objects. `PatientGrid` objects have to be defined differently, because dependencies between objects and their contents in different resources must be considered. In this example we deal with implicit object replicas:

```
create view DoctorGridDef {
virtual_objects DoctorGrid {
//return remote not fragmented objects doc
return (GlobalIndex.Doctor.HFrag.ServerName).Doctor as doc};
//the result of retrieval virtual objects
on_retrieve do {return deref(doc)};
```

```
create view NameDef {
//subobjects Name of object DoctorGrid
virtual_objects Name {return doc.Name as dn};
//the result of retrieval virtual objects

on_retrieve do {return deref(dn)};
};

create view SurnameDef {//?};

create view worksInDef {
virtual_pointers worksIn {return doc.WorksIn as wi};
on_retrieve do {return deref(wi)};
};
};

create view WardGridDef {
virtual_objects WardGrid {
return (GlobalIndex.Ward.HFrag.ServerName).Ward as war};
on_retrieve do {return deref(war)};
create view NameDef {
virtual_objects Name {return war.Name as wn};
on_retrieve do {return deref(wn)};
};

create view IDDef { //?};
};

create view PatientGridDef {
//below procedure integrates all physical 'Patient' objects into complete virtual
objects 'PatientGrid' independent of fragmentation issues, please notice that some
objects may have data replicas which are not known explicitly for a VR creator
(like personal data)
virtual_objects PatientGrid {
return {
//create a bag of identifiers to all distributed Patient objects
bag ((GlobalIndex.Patient.HFrag.ServerName).Patient as patH),
```

```
(GlobalIndex.Patient.VFrag.ServerName).Patient as patV)).
```
//create a list of all PESEL objects from all servers excluding repetitions as uniquePesel
```
((distinct(patH.PESEL) as uniquePesel).
```
//basing on the unique pesel list, for each unique pesel create bags containing references to Patient objects with the current pesel from every remote resource/server accessible as Patients virtual object
```
bag((ref PatH where PESEL = uniquePesel),
(ref PatV where PESEL = uniquePesel))) as Patients };
```
//as a result we get as many Patient objects as instances of unique PESEL objects available from all servers, please notice that some PatientGrid objects can have replicas with personal data objects such Name, Surname, Address, PESEL, moreover to solve the situation where query about Name returns a number names for one PESEL we propose additional procedures of retrieving and calling these objects separately.
```
};
```
//return a complete information about every Patient
```
on_retrieve do { return deref(Patients) };

create view DiseaseDef {
virtual_objects Disease { return distinct(Patients.Disease
as PatDis };
```
//return remote not fragmented Disease objects without repetitions
```
on_retrieve do {return deref(distinct(PatDis))};
};
```
//here should be definitions of access procedures to achieve; Name objects, Surname objects, Medicines objects and PESEL objects that are similar to access procedure of Disease objects

```
create view isAssignedToDef { virtual_pointers isAssignedTo
{return Patients.isAssignedTo as iat};
on_retrieve do {return deref(iat)};
};
};
```

The above example is focused on processing a data schema showed in Figure 5, so it demonstrates only the necessary procedures of integration objects shaped in

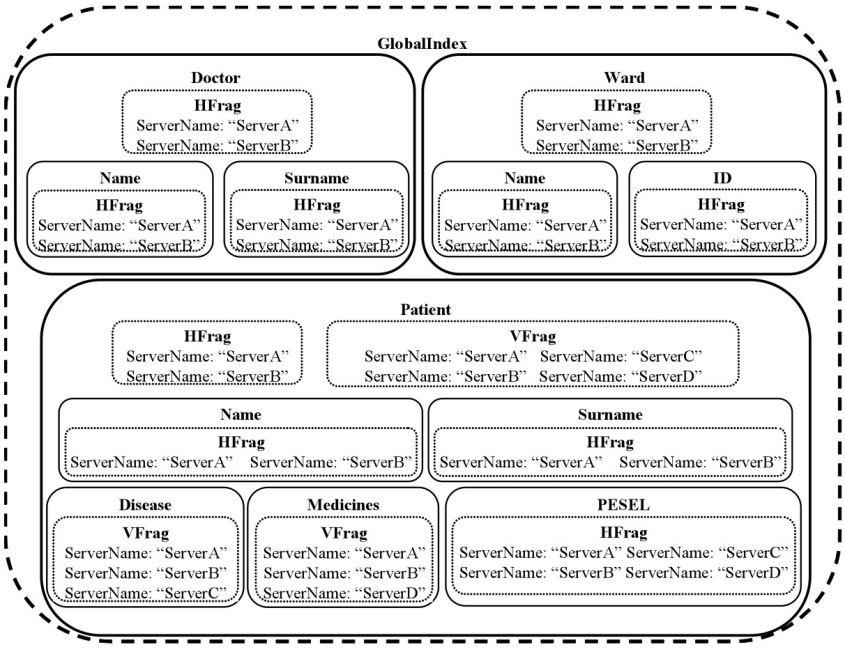

Figure 7. The contents of CMU global index for example of mixed fragmentation

different fragmentation applying to exampled query. Actually, the presented proposal has no limitations in designing a fully automatic integration process by extending the above views with the additional integration routines (like for 'Disease' and 'Name' objects) for every object indexed in the CMU global index.

3.2. Wrapping

Let us consider an object-oriented data model available in the VR (see Figure 5). Its relational reflection available at back-end of the wrappers is presented on Figure 8. For the exemplification clearness we utilize only the most important of its parts, moreover, if we use all its tables, the example will be not acceptable long to show it in this paper. All the table names in relational model are extended by 'R' which means 'relational' to increase the clearness.

Please notice that all relationships between relational tables in OO data schema are realized with virtual pointers like worksIn, isAssignedTo, etc.:

```
   create view DoctorDef {
virtual_objects Doctor {return DoctorR as doc };
virtual_objects Doctor(docID)
{return (DoctorR where ID == docID) as doc };

create view NameDef {
virtual_objects Name {return doc.Name as dn};
on_retrieve do {return dn};
};

create view worksInDef {
virtual_pointers worksIn {return DocToWardR.WardID as w};
on_retrieve do {return Ward(w) as Ward};
};
};

create view WardDef {
virtual_objects Ward {return WardR as war };
virtual_objects Ward(warID)
{return (WardR where ID == warID) as war };

create view NameDef {
virtual_objects Name {return war.Name as wn};
on_retrieve do {return wn};
};
};

create view PatientDef {
virtual_objects Patient {return PatientR as pat };
virtual_objects Patient(patID)
{return (PatientR where PESEL == patID) as pat };
on_retrieve do {return pat};

create view DiseasePatDef {
virtual_objects Disease {return DiseaseR as di};
virtual_objects Disease(disID)
```

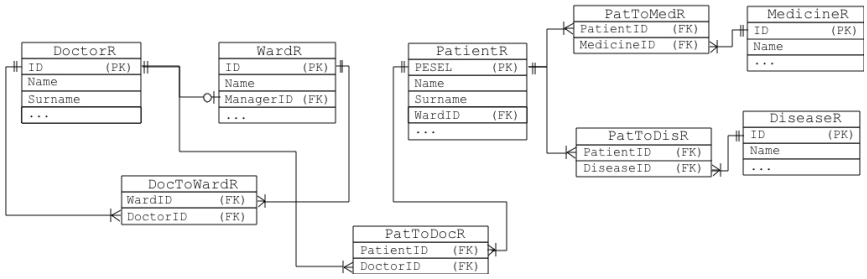

Figure 8. The example of a relational schema, in reflection of OO model in Fig. 5

```
{return (DiseaseR where ID == disID) as di};
on_retrieve do {return (Disease(PatToDisR.DiseaseID)).Name};
};

create view isAssignedToDef {
virtual_pointers isAssignedTo {return PatToDocR.DoctorID as d};
on_retrieve do {return Doctor(d) as Doctor};
};
};
```

Consider a query appearing at the front-end of the wrapper, see example query for integration in subsection 3.1 and some additional information about unique indexes in relational tables (primary-foreign key integrity), the optimization procedures in wrapper are performed by the following steps:

1. Introduce implicit `deref` (dereference) functions:

```
Patient where (deref(Disease) = "cancer" and
isAssignedTo.Doctor.worksIn.Ward.deref(Name) = "cardiology");
```

2. Substitute `deref` with the invocation of `on_retrieve` function for virtual objects and `on_navigate` for virtual pointers:

```
Patient where ((Disease(PatToDisR.DiseaseID)).Name) = "cancer"
and isAssignedTo.(Doctor(d) as Doctor).Doctor.worksIn.Ward(w)
as Ward.Ward.(Name.wn) = "cardiology");
```

3. Substitute all view invocations with the queries from sack definitions:

```
PatientR where ((DiseaseR where ID ==
PatToDisR.DiseaseID as di).Name) = "cancer" and
(PatToDocR.DoctorID as d).(((DoctorR where ID == d) as doc)as
Doctor).Doctor.(DocToWardR.WardID as w).((WardR where ID == w)
as war) as Ward).Ward.((war.Name as wn).wn) = "cardiology");
```

4. Remove auxiliary names `w` and `d`:

```
PatientR where ((DiseaseR where ID ==
PatToDisR.DiseaseID).Name) = "cancer" and
(((DoctorR where ID == PatToDocR.DoctorID) as doc) as
Doctor).Doctor.(((WardR where ID == DocToWardR.WardID)
as war) as Ward).Ward.(war.Name) = "cardiology");
```

5. Remove auxiliary names `doc` and `war`:

```
PatientR where ((DiseaseR where ID ==
PatToDisR.DiseaseID).Name) = "cancer" and
((DoctorR where ID == PatToDocR.DoctorID) as
Doctor).Doctor.((WardR where ID == DocToWardR.WardID as
Ward).Ward.Name = "cardiology");
```

6. Decompose the query on small parts:

```
PatientR where ((DiseaseR where ID ==
PatToDisR.DiseaseID).Name) = "cancer" and
(DoctorR where ID == PatToDocR.DoctorID) and ((WardR where
ID == DocToWardR.WardID) and (WardR.Name = "cardiology")));
```

7. Here unique indexed sub-query can be substituted with `exec_immediately` clause with SQL conversion:

```
PatientR where ((DiseaseR where ID ==
PatToDisR.DiseaseID and Name) = "cancer" and
(DoctorR where ID == PatToDocR.DoctorID) and
(exec_immediately (SELECT * FROM WardR WHERE ID ==
DocToWardR.WardID AND WardR.Name = "cardiology")));
```

8. Because another integrity constraint is available to the wrapper, the pattern is detected and another `exec_immediately` substitution is performed:

```
PatientR where ((exec_immediately (
SELECT * FROM DiseaseR WHERE ID == PatToDisR.DiseaseID AND
DiseaseR.Name = "cancer") and
(DoctorR where ID == PatToDocR.DoctorID) and
(exec_immediately (SELECT * FROM WardR WHERE ID ==
DocToWardR.WardID AND WardR.Name = "cardiology")));
```

Either of the SQL queries invoked by `exec_immediately` clause is executed in the local relational resource and pends native optimization procedures (with application of indices and fast join, respectively).

4. Conclusions and Future Work

In this paper authors presented a complete solution to a transparent integration of distributed data in the Virtual Repository mechanism and the Data Grid architecture. The solution utilizes a consistent combination of several technologies such as; SBA object-oriented database as basic system for design a data grid architecture; the SBA's query language SBQL as an interface for processing distributed data; virtual updatable views as a complex structure for a distributed data virtualization into a global schema; P2P networks developed on the ground of JXTA as a background layer for unlimited communication and finally a distributed index for grid resources and object-oriented to relational wrapper idea plus its architecture and example processing.

A preliminary implementation solves a very important issue of independence between technical aspects of distributed data retrieving through the wrappers, managing through set of views (including additional issues such as participants' incorporation, resource contribution) and a logical virtual repository content scalability (a business information processing). We expect that the presented solution will be efficient and fully scalable. We also expect that due to the power of object-oriented databases and SBQL such a mechanism will be more flexible than other similar solutions (if they appear). The prototype is implemented and preliminarily tested. Currently we are working on extending the presented idea to achieve better flexibility for a real data models. This will include solutions to managing the data replicas and redundancies residing inside distributed resources.

References

[1] Foster, I., Kesselman, C., Nick, J., and Tuecke, S., *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. Global Grid Forum, June 22, 2002.

[2] Habela, P., Kaczmarski, K., Kozankiewicz, H., Lentner, M., Stencel, K., and Subieta, K., *Data-Intensive Grid Computing Based on Updateable Views.* ICS PAS Report 974, June 2004.

[3] Kozankiewicz, H., Stencel, K., and Subieta, K., *Implementation of Federated Databases through Updateable Views.* Proc. EGC 2005 - European Grid Conference, Springer LNCS, 2005.

[4] Kozankiewicz, H., Stencel, K., and Subieta, K., *Integration of Heterogeneous Resources through Updatable Views.* ETNGRID2004 WETICE2004, Proceedings published by IEEE.

[5] Kuliberda, K., Adamus, R., Wislicki, J., Kaczmarski, K., Kowalski, T., and Subieta, K., *Autonomous Layer for Data Integration in a Virtual Repository.* International Conference on Grid computing, high-performAnce and Distributed Applications (GADA'06), Springer 2006 LNCS 4276, pp. 1290-1304.

[6] Kuliberda, K., Kaczmarski, K., Adamus, R., Błaszczyk P., Balcerzak, G., and Subieta, K., *Virtual Repository Supporting Integration of Pluginable Resources.* 17th DEXA 2006 and 2nd International Workshop on Data Management in Global Data Repositories (GRep) 2006, Proc. in IEEE Computer Society.

[7] Kuliberda, K., Wislicki, J., Adamus, R., and Subieta, K., *Object-Oriented Wrapper for Relational Databases in the Data Grid Architecture.* OTM Workshops 2005, Springer LNCS 3762, 2005, pp. 367-376.

[8] Kuliberda, K., Wislicki, J., Adamus, R., and Subieta, K., *Object-Oriented Wrapper for Semistructured Data in a Data Grid Architecture.* 9th International Conference on Business Information Systems 2006, LNI vol. P-85, GI-Edition 2006, pp. 528–542.

[9] Litwin, W., Nejmat, M. A., and Schneider, D. A., *LH*: Scalable, Distributed Database System.* 1996. ACM Trans. Database Syst., 21(4) pp. 480-525.

[10] Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmer, M., and Risch, T., *EDUTELLA, a P2P networking infrastructure based on RDF.* Proc. Intl. World Wide Web Conference, 2002.

[11] *Open Grid Services Architecture, Data Access and Integration Documentation*, http://www.ogsadai.org.uk

[12] Plodzien, J., *Optimization Methods in Object Query Languages*, PhD Thesis. IPIPAN, Warsaw 2000.

[13] *Project JXTA Community: http://www.jxta.org (home-page)*

[14] Subieta, K., *Stack-Based Approach (SBA) and Stack-Based Query Language (SBQL).* http://www.ipipan.waw.pl/subieta, Description of SBA and SBQL, 2006

[15] Subieta, K., *Theory and Construction of Object-Oriented Query Languages.* Editors of the Polish-Japanese Institute of Information Technology, 2004 (in Polish)

[16] Tatarinov, I., Ives, Z., Madhavan, J., Halevy, A., Suciu, D., Dalvi, N., Dong, X., Kadiyska, Y., Miklau, G., and Mork, P., *The Piazza Peer Data Management Project*, ACM SIGMOD Record, Vol. 32, No. 3, 2003.

[17] Tatarinov, I. and Halevy, A., *Efficient Query Reformulation in Peer Data Management System*, SIGMOD Conference 2004, pp. 539–550.

[18] Wilson, B., *JXTA Book, http://www.brendonwilson.com/projects/jxta/ (home-page)*

[19] Kozankiewicz, H., *Updateable Object Views.* PhD Thesis, 2005, http://www.ipipan.waw.pl/subieta/

[20] Lentner, M. and Subieta, K., *ODRA: A Next Generation Object-Oriented Environment for Rapid Database Application Development*, 2006, http://www.ipipan.waw.pl/subieta/artykuly/ODRA paperpl.pdf

[21] Plodzien, J. and Krakeni, A., *Object Query Optimization in the Stack-Based Approach.* Proc. ADBIS Conf., Springer LNCS 1691, pp. 303-316, 1999.